FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico Final
Jazz Jackrabbit 2

[75.42] Taller de Programación I
Primer Cuatrimestre de 2024

Estudiante	**Padrón**	**Email**
Buono, Fernando	103523	fbuono@fi.uba.ar
Duca, Francisco	106308	fduca@fi.uba.ar
Oshiro, Lucas	107024	loshiro@fi.uba.ar
Shiao, Tomás Jorge	106099	tshiao@fi.uba.ar

Índice

1. Introducción

El proyecto es una implementación del juego Jazz Jackrabbit 2 con soporte multijugador. Los jugadores pueden seleccionar uno de los tres personajes: Jazz, Spaz o Lori, cada uno con habilidades y ataques especiales únicos. El juego incluye la capacidad de disparar, correr, saltar y realizar ataques especiales, así como varios estados como intoxicado, recibir daño y muerte. Además, se incluye un editor de niveles para que se puedan crear nuevos niveles.

2. Estructura del Proyecto

El proyecto tiene una estructura de cliente servidor, donde el servidor es el encargado de manejar la lógica del juego y el cliente es el encargado de mostrar la información al usuario. Además, se incluye un editor de niveles que permite crear nuevos niveles para el juego.

2.1. dependencias

1. CMake: para la compilación del proyecto.
2. Qt5: para la interfaz gráfica, versión 5.15.
3. YAML-cpp: para el manejo de archivos de configuración.
4. SDL2: para la interfaz gráfica.
5. SDL2pp: para la interfaz gráfica.
6. libfmt

En caso de usar Vagrant:

1. Vagrant
2. VirtualBox

2.2. Common

Los archivos en común que usan tanto cliente como servidor son:

1. Socket
2. Thread
3. Queue
4. DTO: esto incluye todos los tipos de DTO que se envían entre el cliente y el servidor.
5. Types: esto incluye todos los tipos de datos que se utilizan en el juego.
6. Manejador de mapas
7. Configuración: esto son las clases que procesan los archivos de configuración.

2.3. Servidor

Dada la necesidad de manejar múltiples conexiones, el servidor tiene una estructura que soporta múltiples hilos.

Al iniciar, el servidor lanza un hilo aceptador, que se encuentra constantemente escuchando el puerto. Cuando el cliente se une, se acpeta esa conexión y se crea a la entidad del jugador, que tiene en sí, sus propios hilos emisor y receptor, así también las colas de mensajes recibidos y a enviar.

Cuando llega un mensaje, este es deserialiado por el deserializador. Una vez deserializado, es ejecutado por los manejadores de los comandos, que dependiendo del tipo de DTO realiza las acciones correspondientes.

El último hilo que se tiene es el gameloop, que es quien se encarga de actualizar el juego y enviar los mensajes a los clientes. Esto lo hace recibiendo en cada loop las actualizaciones del juego correspondientes y enviádnolas a todos los clientes, es decir, haciendo un broadcast.

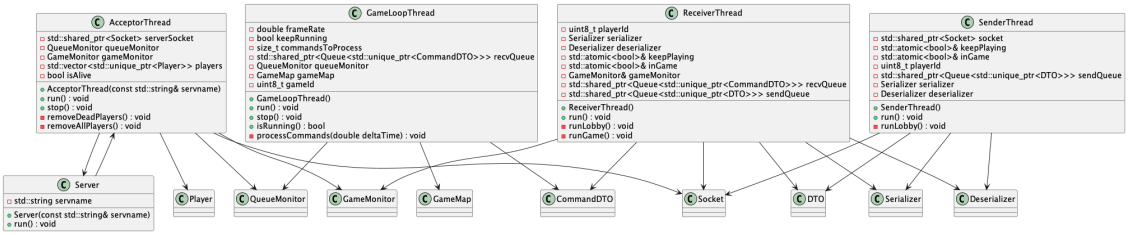

Figura 1: Diagrama de Clases de los hilos del servidor.

Siendo una estrctura multithreading, se tiene que tener en cuenta la concurrencia. Para ello, se utilizan monitores, que son dos: `GameMonitor` y `QueueMonitor`. Ambos utilizan mutex para asegurar que no haya problemas de concurrencia ni race conditions en las actualizaciones de los juegos o procesando comandos. GameMonitor es quien se encarga de actualizar el juego, mientras que QueueMonitor es quien agrega a los clientes a los juegos o realiza un broadcast para informar acerca de una actualización en la partida.

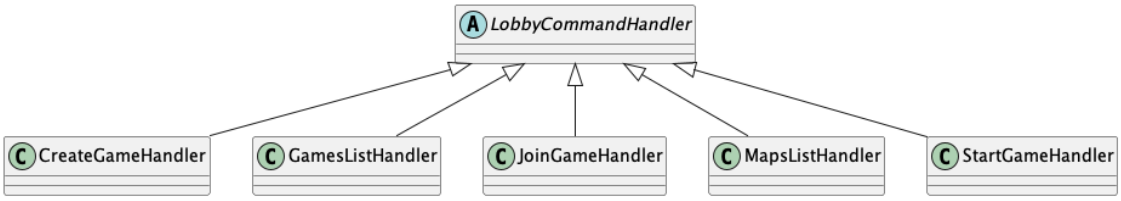

Figura 2: Lobby Command Handlers.

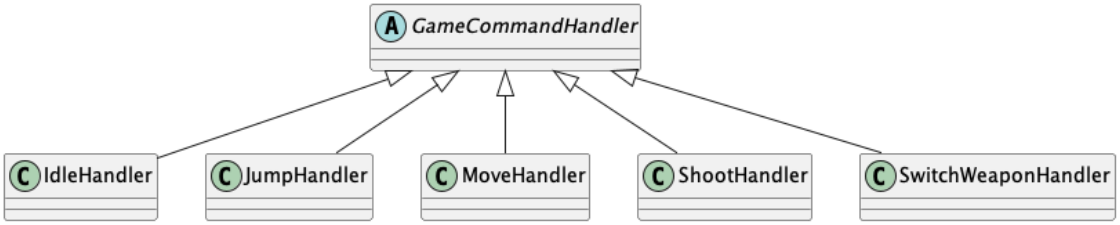

Figura 3: Game Command Handlers.

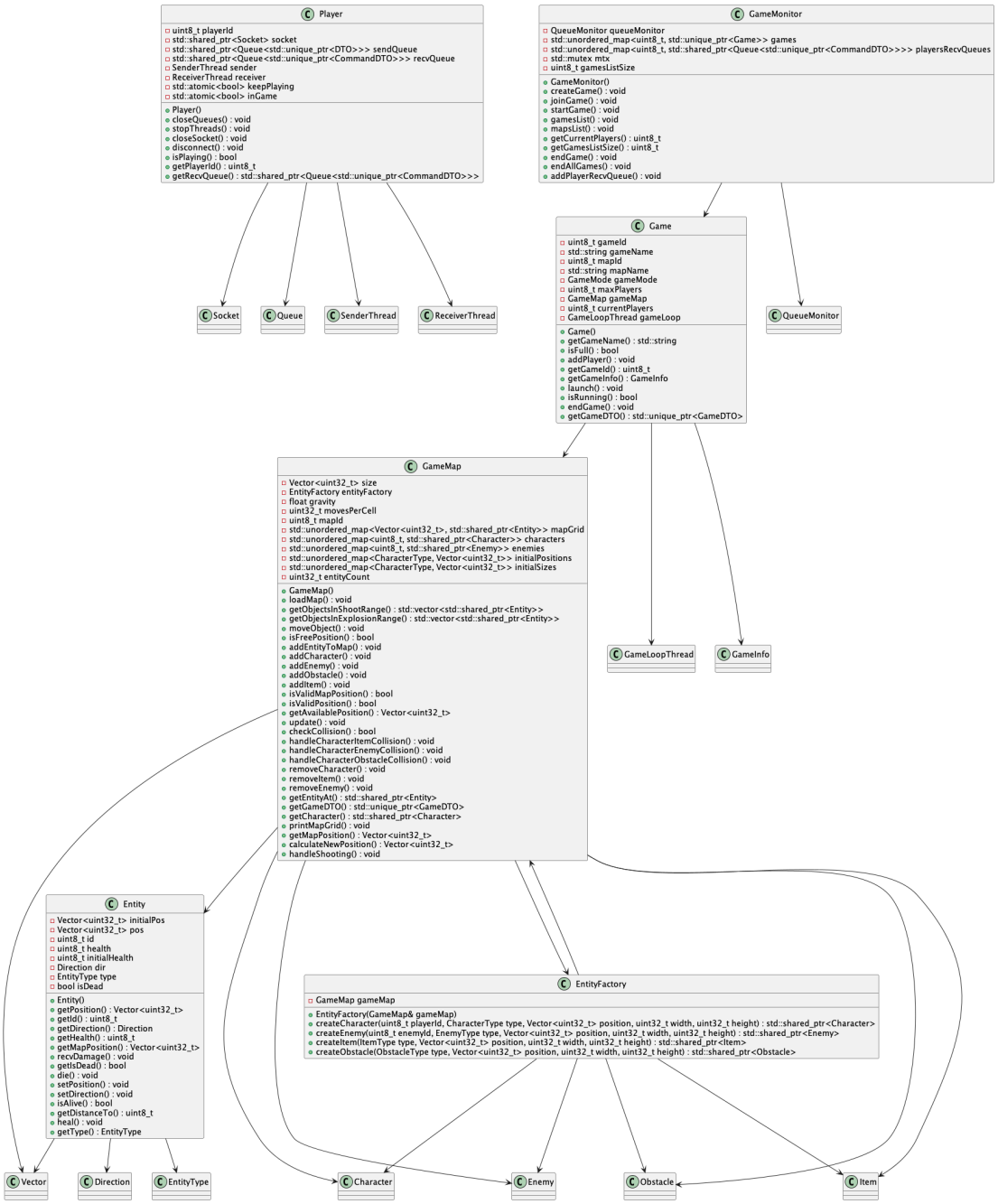

Figura 4: Diagrama de Clases del Modelo del Juego, lado servidor.

2.4. Cliente

El cliente posee una estructura simple que se puede dividir en dos grandes áreas: gameplay y lobby. El gameplay es la parte del juego donde se muestra el mapa y los jugadores pueden interactuar entre sí. El lobby es la parte del juego donde los jugadores pueden seleccionar su personaje y unirse a una partida.

Ambas partes necesitan enviar y recibir datos desde el servidor. Es por ello que se tienen dos threads, emisor y receptor, que se encargan de enviar y recibir datos respectivamente. Es el cliente mismo quien se encarga de lanzar ambos hilos y también de frenarlos.

Para lograr la comunicación con el servidor, también se poseen tres colas: una para el hilo emisor, donde se encolan los mensajes a enviar; en cuanto al receptor, tiene dos: una para los mensajes recibidos que sean del Lobby y otra para aquellos del Gameplay.

Antes de ser enviados, los mensajes son serializados, y antes de ser enviados a sus respectivas colas, son deseralizados. Esto se logra a partir de las clases `Serializer` y `Deserializer`.

La clase `Protocolo` es quien realmente conoce cómo es el protocolo de recepción, este es quien realmente procesa la recepción de datos.

Por último, se tienen los controladores: `LobbyController` y `GameController`. Estos tienen a la cola de mensajes recibidos de cada uno y van desencolandolos y, en función del mensaje recibido, realizan las acciones correspondientes.

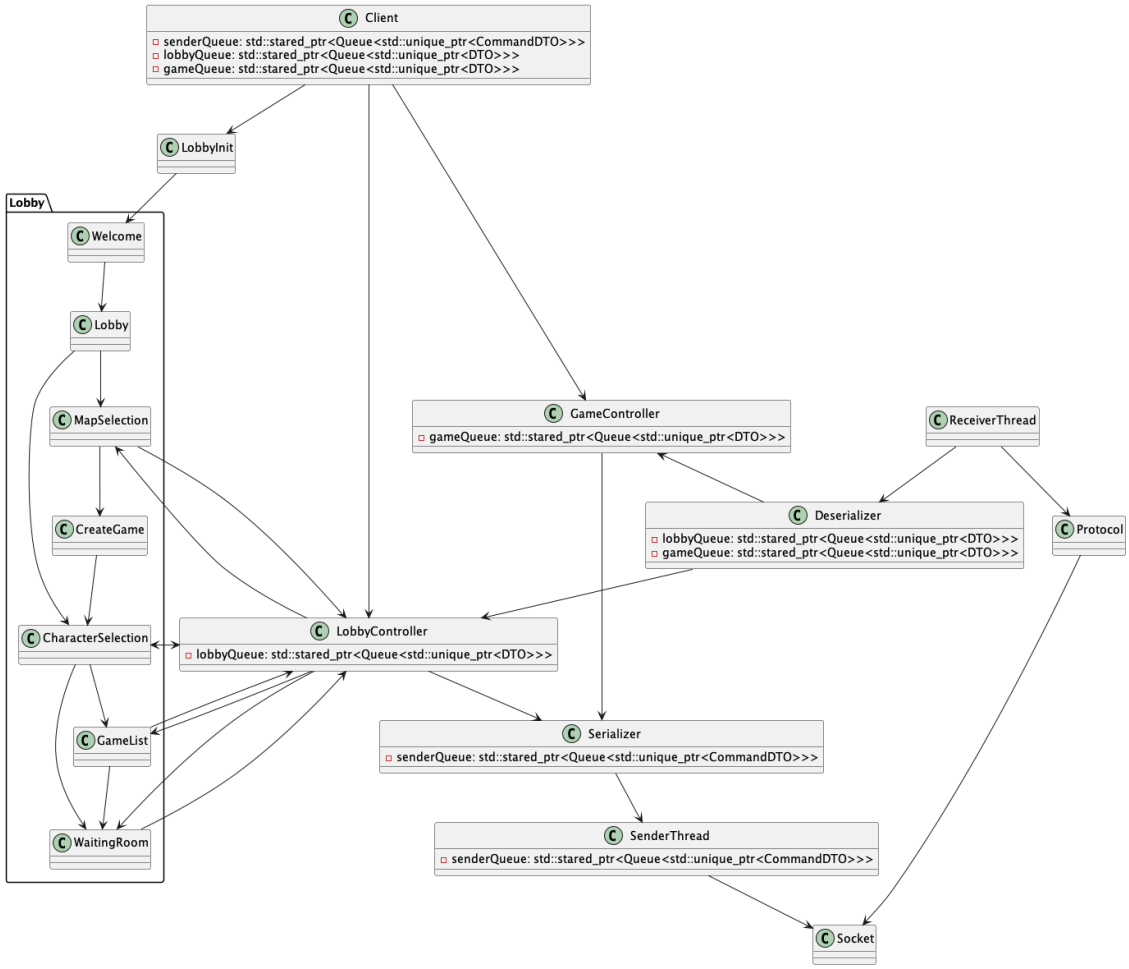

Figura 5: Diagrama de Clases del Lobby.

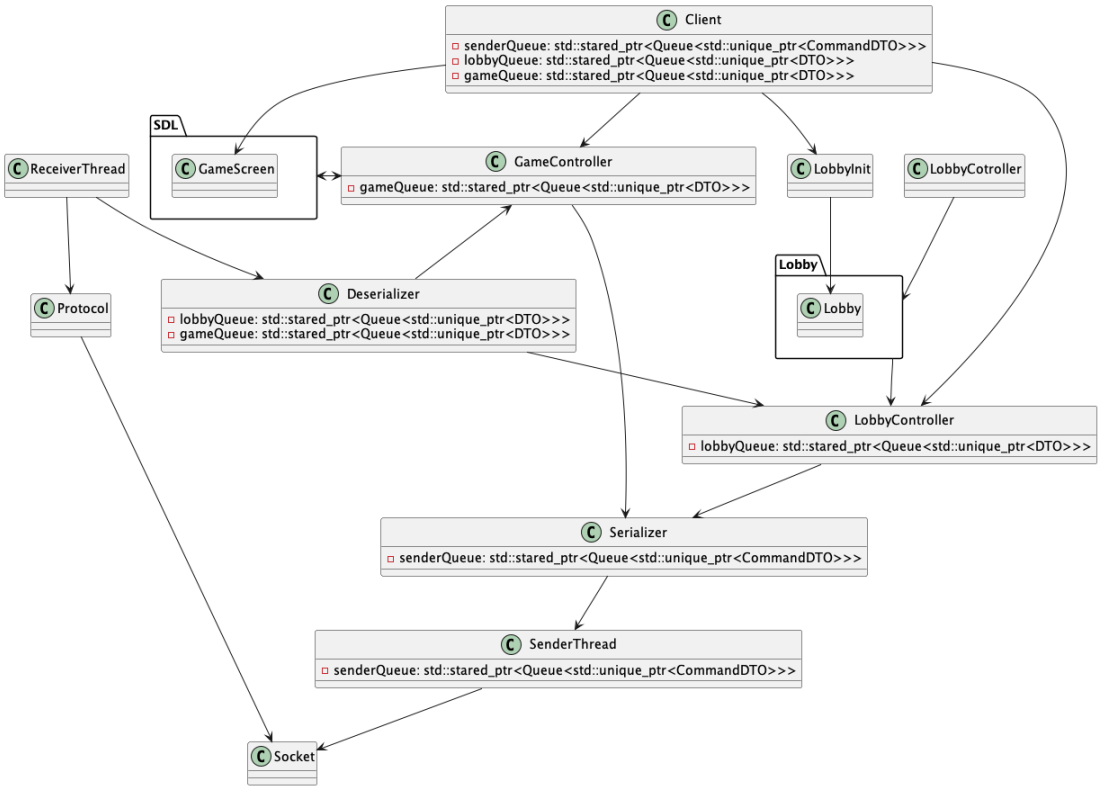

Figura 6: Diagrama de Clases del lado cliente.

2.5. Editor de Niveles

El editor de niveles es un programa simple que consiste de una sola ventana. En ella, se encuetran un canvas, representando el mapa del juego, y a derecha una lista de elementos que se pueden arrastrar sobre el canvas. De esta manera, se pueden crear nuevos niveles para el juego.

Para obtener cada uno de los elementos, se tiene una lista dentro de la clase, asociada a la categoría y al nombre del elemento. Esto es así debido al formato con el que se crea el mapa, donde el YAML tiene la siguiente estructura:

```
CATEGORIA1:
  ELEMENTO_CATEGORIA1_1: [...]
  ...
  ELEMENTO_CATEGORIA1_N: [...]
...
CATEGORIAN:
  ELEMENTO_CATEGORIAN_1: [...]
  ...
  ELEMENTO_CATEGORIAN_M: [...]
```

Listing 1: Ejemplo de formato YAML del mapa

Los elementos también están asociados a Sprites, que son las imágenes que se pegan sobre el mapa. Para ello, se tiene un archivo YAML donde se tiene por cada elemento, la fuente de la imagen, las coordenadas y el ancho y alto que debe tener el sprite en el mapa real. Al necesitar una, se posee una clase `SpritesManager`, que obtiene y renderiza las imágenes.

El canvas está compuesto por un `QGridLayoutWidget` que contiene un `QGridLayout`. Se le crea a ese elemento perteneciente a la clase `LevelEditor`, la venta principal, un `CanvasWidget`, que

acepta que se le arrastren y suelten los elementos encima. Cada vez que se suelte un elemento encima de ella, se crea un nuevo `DroppedElement` y se guarda la posición en la que cayó, en coordenadas del mapa real.

Figura 7: Diagrama de Clases del Editor de Niveles

3. Protocolo

El protocolo que se diseñó para este Trabajo Práctico utiliza Data Transfer Objects (DTO) para enviar y recibir mensajes tanto desde el lado cliente como del lado servidor.

Se crea un DTO por cada tipo de mensaje: Crear Partida, Unirse a una partida, Obtención de los mapas disponibles, Obtención de las partidas disponibles, Actualización de la partida, Actualización del juego, así como otros que contienen datos acerca de los elementos del juego: jugador, bala, enemigo, ítem, etc.

Cada DTO contiene datos específicos que se deben enviar hacia el otro lado, previamente serializados por el serializador. En la otra punta, el deserializador se encarga de deserializar el mensaje y convertirlo en un DTO para ser usado.

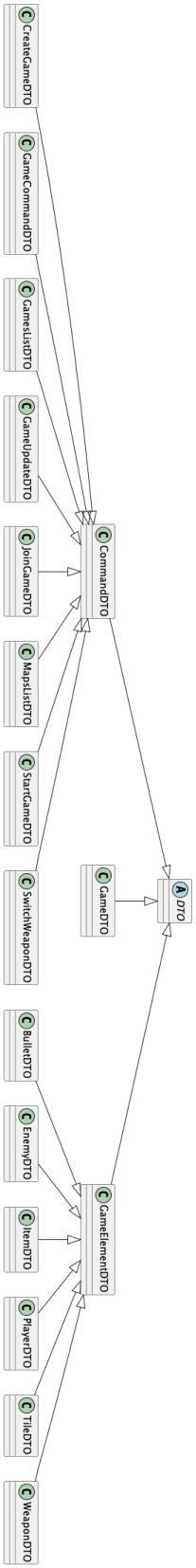

Figura 8: Jerarquía de los Data Travel Objects.

4. Vagrant e Instalador

Se provee un archivo `Vagrantfile` que permite crear una máquina virtual con todas las dependencias necesarias para compilar y ejecutar el proyecto. Para ello, se debe tener instalado Vagrant y VirtualBox.

El archivo crea una máquina virtual utilizando la box `jammy64`, que utiliza el sistema operativo Ubuntu 22.04 y contiene todas las dependencias que se precisan para compilar y ejecutar el proyecto. Ejecutando `vagrant up -provision`, se crea la máquina virtual y se instalan las dependencias. Luego, se puede acceder a la máquina virtual con `vagrant ssh` y compilar y ejecutar el proyecto.

5. Compilación y Ejecución

Para compilar el proyecto, se debe ejecutar los siguientes comandos en la terminal sobre el directorio del proyecto:

```
$ sudo rm -rf build # Si es que existe una carpeta build previa
$ mkdir -p build
$ cd build
$ cmake ..
$ make -j$(nproc) # Alternativamente, make -j4
```

Listing 2: Compilación del Proyecto

Una vez compilado, para ejecutar el servidor se debe correr:

```
$ ./jazzserver <port> # Reemplazar <port> por un puerto a elección
```

Listing 3: Ejecución del Servidor

Para ejecutar el cliente, se debe ejecutar:

```
$ ./jazzclient <ip> <port> # Reemplazar <ip> por la IP del servidor (localhost) y
    <port> por el puerto
```

Listing 4: Ejecución del Cliente

Por último, para el editor de niveles:

```
$ ./levelеditor
```

Listing 5: Ejecución del Editor de Niveles

5.1. Utilizando Vagrant

Se provee también un archivo `Vagrantfile` que permite crear una máquina virtual con todas las dependencias necesarias para compilar y ejecutar el proyecto. Para ello, se debe tener instalado Vagrant y VirtualBox.

Para crear la máquina virtual y correr el proyecto, en la carpeta donde se bajó el código del repositorio, se debe ejecutar:

```
$ vagrant up --provision
$ vagrant ssh
$ cd /home/vagrant/jazz_jackrabbit_2/build
$ ./jazzserver <port> # Para correr el servidor, reemplazar <port> por un puerto a
    elección
```

```
$ ./jazzclient <ip> <port> # Para correr el cliente, reemplazar <ip> por la IP del
    servidor (localhost) y <port> por el puerto
$ ./levelеditor # Para ejecutar el editor de niveles
```

Listing 6: Creación de la Máquina Virtual

El archivo `Vagrantfile` se encarga de instalar las dependencias necesarias y compilar el proyecto. Para correr el servidor y el cliente, se debe ejecutar el servidor primero y luego el cliente.

Para salir de Vagrant, una vez finalizada la ejecución:

```
$ exit
$ vagrant halt
$ vagrant destroy # Si se desea borrar la máquina virtual
```

Listing 7: Salir de Vagrant